CP: 06/09/2022
DP: 06/09/1999
Day: Tuesday

Lecture 70 : Exceptional Handling : (54 pages)

Theory
1. Introduction
2. Runtime stack mechanism.
3. Default Exceptional Handling in Java.
4. Exception Hierarchy.
5. Customized exception Handling by using try catch.
6. Control flow in try catch.
7. Method to print exception information.
8. try with multiple catch blocks.
9. finally block.
10. difference Bfw final, finally, finalize.
11. Control flow in try catch finally.
12. Control flow in nested try-catch-finally.
13. Various possible combinations of try catch finally.
14. throw keyword.
15. throws keyword.
16. Exceptional Handling Keywords Summary.
17. Various possible Compile time errors in exception handling.
18. Customized or user defined exceptions.
19. Top-10 exceptions.
20. 1.7 version enhancements
    1. try with resources.
    2. multi-catch block.

1. **Introduction :** An un-expected unwanted event that disturbs normal flow of the program, is called exception.

eg:- TyrePunctured Exception, sleeping Exception, File Not Found Exception etc.

It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of the program.

Exception Handling doesn't mean repairing an exception, we have to provide alternative way to continue rest of the program normally. is the concept of exception handling

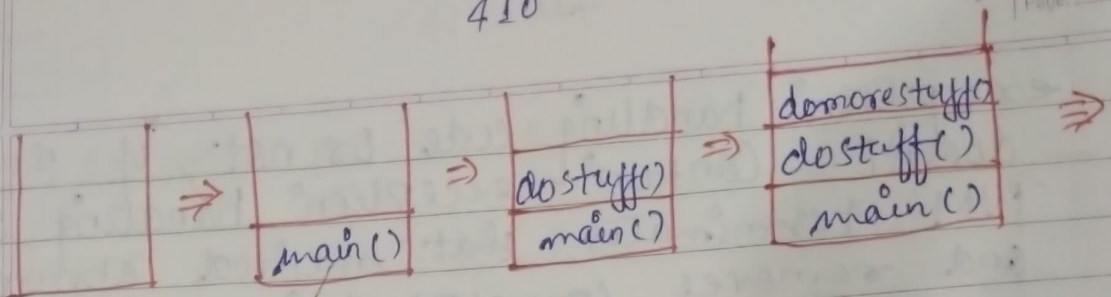
for eg:- Our programing requirement is to read data from remote file locating at London, at runtime if London file is not available our program should not be terminated abnormally, we have to provide some local file To continue rest of the program normally this way of defining alternative is nothing but exception handling.

eg:-
```
try
{
    read data from remote file locating at London.
}
catch (FileNotFoundException e)
{
    use local file and continue rest of the Program normally.
}
```

2. Runtime stack mechanism :- 1. For every thread JVM will create a Runtime stack.
2. Each & every method call performed by the thread will be stored in the corresponding stack.
3. Each entry in the stack is called stack frame or activation record.
4. After completing every method call the corresponding entry from the stack will be removed.
5. After completing all the method calls the stack will become empty and that empty stack will be destroyed by JVM, just before terminating the thread.

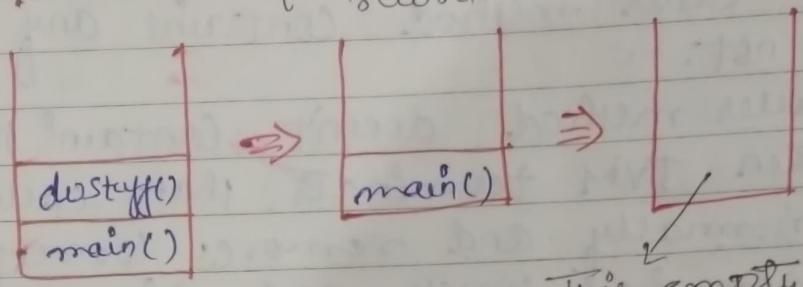
```
class Test
{
    p s v main(String[] args)
    {
        dostuff();
    }
    p s v dostuff()
    {
        domore.stuff();
    }
    p s v domore.stuff()
    {
        sop("Hello");
    }
}
```

op: Hello



Runtime Stack for main thread

Stack frame/ Activation record.



This empty stack will be destroyed by JVM.

Lecture 71 :- (3) Default Exception Handling in Java :-

1- Inside a method if any exception occurs the method in which it is rise is responsible to create exception Object by including the following information.

1 - Name of Exception
2 - Description of exception
3 - location at which exception occurs [Stack Trace].

2- After creating exception Object method handover that objects to the JVM.

3- JVM will check whether the method contains any

exception handling code or not, if the method doesn't contain exception handling code, then JVM terminates that method abnormally and removes corresponding entry from the stack.

4- Then JVM identifies caller method and checks, whether caller method contains any handling code or not.

5- If the caller method doesn't contain handling code, then JVM terminates that caller method also abnormally and removes corresponding entry from the stack and this process will be continued until main method and if the main method also doesn't contain handling code then JVM terminates main method abnormally and removes corresponding entry from the stack.

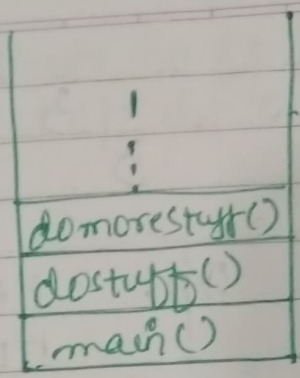
6- Then JVM handovers responsibility of exception handling to default exception handler, which is the part of JVM.

7 default exception handler prints exception information in the following format and terminates program abnormally.

Exception in thread "main/xxx": Name Of Exception: Description Stack trace.

Eg:
```
class Test
{
    p s v main(String[] args)
    {
```

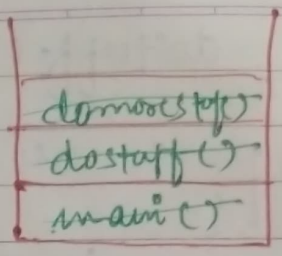
```
    dostuff();
}
P.s.v dostuff()
{
    doMorestuff();
}
P.s.v doMorestuff()
{
    sop(10/0);
}
```




```
Exception in thread "main" j.l.AE: / by zero
    at Test.doMorestuff()
    at Test.dostuff()
    at Test.main().
```

eg: class Test
{
    P.s.v main(String[] args)
    {
        dostuff();
        sop(10/0);
    }
    P.s.v dostuff()
    {
        doMorestuff();
        sop("Hi");
    }
    P.s.v doMorestuff()

sopln ("Hello");
3
3



O/p: Hello
Hi
Exception in thread "main" j.l.AE: / by zero.
at Test.main()

Note: 1. In a program if atleast one method terminates abnormally, then the program termination is abnormal termination.
2. If all methods terminated normally then only program termination is normal termination.

Imp 4. Exception Hierarchy:

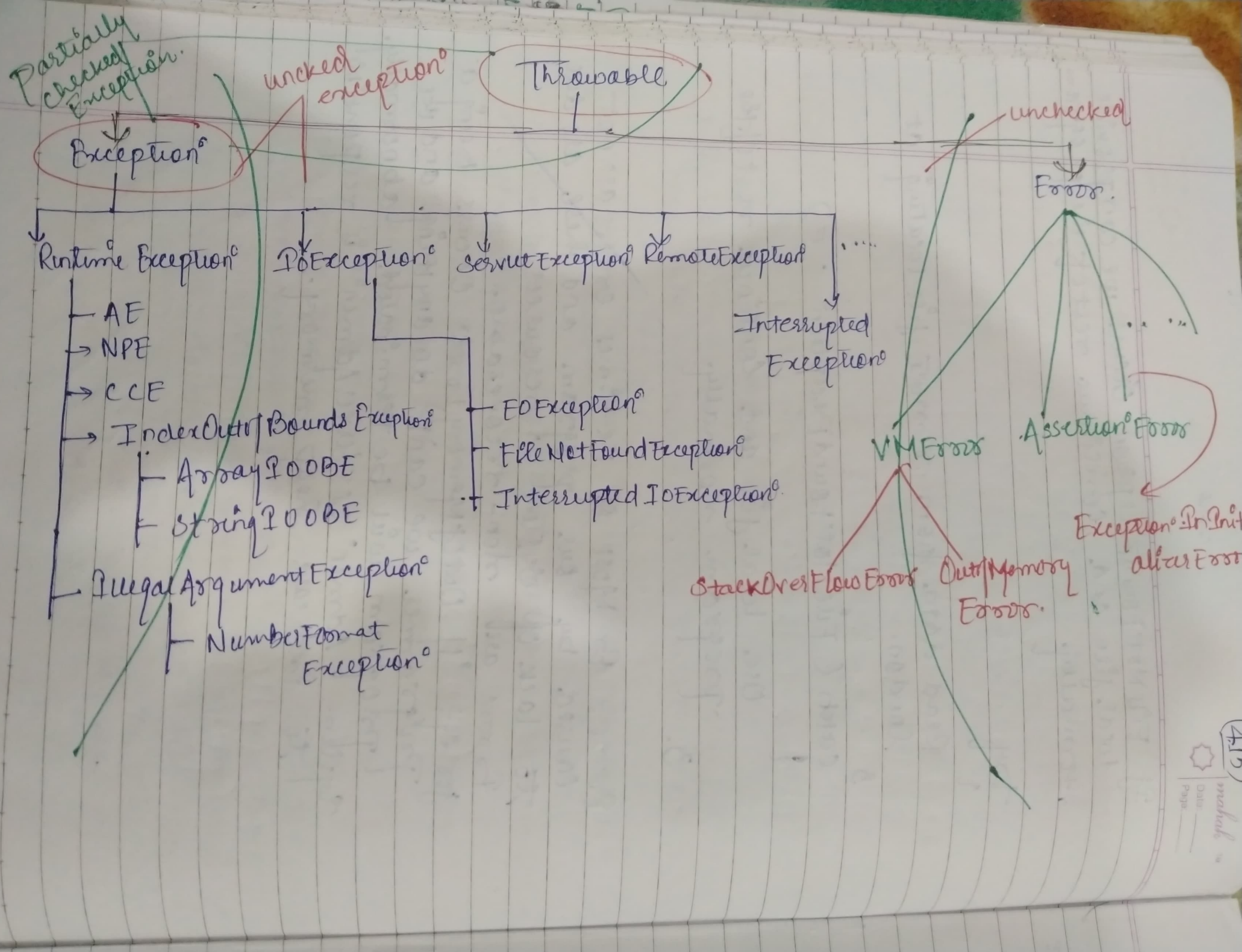
Throwable class act as root for Java exception Hierarchy.
Throwable class defines two child classes -
1. Exception
2. Error.

1. Exception: Most of the times exceptions are caused by our program and these are recoverable.
for eg: if our programming requirement is to read data from remote file locating at London. at runtime if remote file is not available then we will get RTE: FileNotFoundException.

If FileNotFoundException occurs we can provide local file and continue rest of the program normally.

```
try
{
    Read data from remote file locating at
    london.
}
catch (FileNotFoundException e)
{
    Use local file and continue rest of the
    program normally.
}
```

Error :- Most of the times errors are not caused by our program and these are due to lack of system resources.
Errors are non-recoverable.
for eg :- If OutOfMemoryError occurs being a Programmer we can't do anything. and the program will be terminated abnormally.
System Admin or ServerAdmin is responsible to increase heap memory.


Partially checked Exception.
unchecked exception
Throwable
Exception
unchecked
Error.
RuntimeException
IOException
ServletException
RemoteException
InterruptedException
AE
NPE
CCE
IndexOutOfBoundsException
ArrayIOOBE
StringIOOBE
IllegalArgumentException
NumberFormatException
EOException
FileNotFoundException
InterruptedIOException
VMError
AssertionError
StackOverFlowError
OutOfMemoryError.

Lecture 72 :-

Checked Exceptions Vs Unchecked Exceptions :-

Checked Exception :-
The Exceptions which are checked by compiler for smooth execution of the program at runtime are called checked exception. eg-
HallTicketMissingException.
PenNotWorkingException.
FileNotFoundException.

In our program if there is a chance of rising checked exception then, compulsory we should handle that checked exception [either by try catch or by throws keyword] otherwise we will get CTE.

Unchecked Exception :- The exceptions which are not checked by compiler whether programmer handling or not such type of exceptions are called unchecked exceptions eg:- ArithemeticException, BombBlastException etc.

Note :- 1) Whether it is checked or unchecked every exception occurs at runtime only there is no chance of occuring any exception at CT.

(2) RuntimeException and its child classes, Error and its child classes are unchecked. except these remaining are checked.

* Fully checked Vs partially checked :-

A checked Exception is said to be fully checked iff & all its child classes also checked.
eg:- IOException, InterruptedException.

A checked Exception is said to be partially checked iff some of its child classes are unchecked. eg:- Exception, Throwable.

Note:- The only possible partially checked Exceptions in Java are —
1) Exception.
2) Throwable

Describe the behaviour of following exceptions?

1) IOException — checked (Fully)
2) RuntimeException — Unchecked
3) InterruptedException — checked (Fully)
4) Error → Unchecked
5) Throwable → checked (Partially)
6) ArithmeticException. → Unchecked
7) NullPointerException. → Unchecked
8) Exception → checked (Partially)
9) FileNotFoundException. → checked (Fully).

Lecture 79 :

5. Customized Exception handling by using try catch :

It is highly recommended to handle exceptions. The code which may raise an exception is called risky code. and we have to define that code inside try-block. and corresponding handling code we have to define inside catch block.

```
try
{
   Risky code
}
catch (Exception e)
{
   Handling code
}
```

Without try-catch

```
class Test
{
   p s v main(String[] args)
   {
      sopln("stm 1");
      sopln(10/0);
      sopln("stm 3");
   }
}
```

With try-catch.

```
class Test
{
   p s v main(String[] args)
   {
      sopln("stm 1");
      try
      {
         sopln(10/0);
      }
      catch (ArithmeticExcept e)
```

```
{
    soptn (10/2)
}
soptn ("Stm 3");
```

O/p: Stm 1
RE: AE: / by zero
Abnormal Termination

O/p: Stm 1
5
Stmt 3
Normal Termination.

6. Control flow in° try-catch :-

```
try
{
  Stmt 1;
  Stm 2;
  Stmt 3;
}
Catch (X  e)
{
   Stmt 4;
}
  Stm 5;
```

Case 1:- If there is no exception :-
1, 2, 3, 5, Normal Termn.

Case 2:- If an exception° rise at statement 2 and corresponding catch block matched.
1, 4, 5, Normal Termn

Case 3:-

Case 4:-

Note

Case3: If an exception rise at statement 2, and corresponding catch block not matched.
1, AT.

Case4: If an exception rise at statement 4. or statement 5. then it is always abnormal termination.

Note: 1) Within the try block if anywhere an exception rise then rest of the try block won't be executed even though we handled that exception. Hence within the try block we have to take only risky code and length of try block should be as less as possible.

2) In addition to try block there may be a chance of rising an exception inside catch and finally blocks.

3) If any statement which is not part of try block and rises an exception then it is always abnormal termination.

7. Methods to print Exception information:

Throwable class defines the following methods to print exception information.

Method	printable format.
① printStackTrace()	Name of Exception: Description StackTrace.

② toString() | Name of Exception : Description
③ getMessage() | Description.

```
class Test
{
    p s v main(String[] args)
    {
        try
        {
            sopln(10/0);
        }
        catch(ArithmeticException e)
        {
            e.printStackTrace();
            sopln(e); (or) sopln(e.toString());
            sopln(e.getMessage());
        }
    }
}
```

java.lang.AE: / by zero at Test.main()

java.lang.AE: / by zero

/ by zero

Internally default Exception handler will use printStackTrace method to print exception information to the console.

8. try with multiple catch blocks:-

The way of handling an exception is varied from exception to exception. Hence for every exception type it is highly

recommended to take seperate catch block.
i.e. try with multiple catch block is always possible and recommended to use.

Worst way

```
try
{
Risky code
}
catch (exception e)
{
}
```

worst programming practice.

Best way.

```
try
{
Risky code
}
catch (ArithmeticException e)
{
perform alternative arithmetic operation.
}
catch (SQLException e)
{
use mySQL db instead of oracle db.
}
Catch (FileNotFoundException e)
{
use local file instead of remote file.
}
Catch (Exception e)
{
// default exception-handling.
}
```

Best programming practice.

If try with multiple catch blocks present then the order of catch blocks is very important. we have to take child first and then parent. Otherwise we will get CTE: Exception xxxx has already been caught.

```
try
{
  Risky code.
}
catch (Exception e)
{
}
catch (ArithmeteeException e)
{
}
```

CE: exception 'j.l.AE has already been caught.

```
try
{
  Risky code
}
Catch(ArithmExcept" e)
{
}
catch (Except" e)
{
}
```

We can't declare two catch blocks for the same exception. Otherwise we will get CTE:

```
try
{
  Risky code
}
catch (AE e)
{
}
catch (AE e)
{
}
```

exception 'j.l.AE has already been caught.

Lecture - 74 :-

9. finally block :-
difference

(10) (a) Final :- final is a modifier applicable for classes, methods and variables.

1) If a class declared as final then we can't extend that class i.e. we can't create child class for that class i.e. inheritance is not possible for final classes.

2) If a method is final then we can't overrede that method in the child class.

3) If a variable declared as final then we can't perform reassignment for that variable.

(9) (2) finally :- finally is a block always associated with try catch to maintain cleanup code.

```
try
{
  Risky code
}
catch (Exception e)
{
  Handling code.
}
finally
{
  cleanup code
}
```

The speciality of finally block is it will be executed always irrespective of whether Exception is rised or not rise and whether handle or not handle.

(3) finalize() :- finalize is a method always invoked by garbage collector just before destroying an object to perform cleanup activities.
Once finalize method completes immediately garbage collector destroys that object.

Note :- finally block is responsible to perform cleanup activities related to try block. i.e. whatever resources we opened as the part of try block will be closed inside finally block.

whereas finalize method is responsible to perform cleanup activities related to object. i.e. whatever resources associated with object will be deallocated before destroying an object by using finalize method.

13. Various possible combinations of try, catch finally.

1) In try - catch - finally order is important.
2) Whenever we are writing try compulsory we should write either catch or finally

Otherwise we will get CTE. i.e. try without catch or finally is invalid.

3) Whenever we are writing catch block, compulsory try block must be required. i.e. catch without try is invalid.

4) Whenever we are writing finally block compulsory we should write try block. i.e. finally without try is invalid.

5) Inside try, catch and finally blocks we can declare try, catch and finally blocks i.e. nesting of try, catch, finally is allowed.

6) for try catch and finally blocks curly braces are mandatory.

① try { } catch (x e) { } ✓

② try { } catch (x e) { } catch (y e) { } ✓

③ try { } catch (x e) { } catch (x e) { } ✗
CE: exception x has already been caught.

④ try { } catch (x e) { } finally { } ✓

⑤ try { } finally { } ✓

(6)
```
try
{
3
catch(X e)
{
3
try
{
3
catch(Y e)
{
3
```
✓

(7)
```
try
{
3
catch(X e)
{
3
try
{
3
finally
{
3
```
✓

(8)
```
try
{
3
```
CE! try without catch (or) finally.

(9)
```
catch(X e)
{
3
```
CE! catch without try.

(10)
```
finally
{
3
```
CE! finally without try.

(11)
```
try
{
3
finally
{
3
catch(X e)
{
3
```
CE! catch without try.

(12)
```
try
{
3
Sop("Hello");
catch(X e)
{
3
```
CE1: try without catch or finally
CE2: catch without try.

(13)
```
try
{
}
catch(X e)
{
}
Sop(n("Hello");
catch(X e)
{
}
```
CE: catch without try.

(14)
```
try
{
}
catch(X e)
{
}
Sop(n("Hello");
finally
{
}
```
CE: finally without try.

(15)
```
try
{
  try
  {
  }
  catch(X e)
  {
  }
}
catch(X e)
{
}
```
✓

(16)
```
try
{
  try
  {
  }
}
catch(X e)
{
}
```
CE: try without catch or finally.

(17)
```
try
{
  try
  {
  }
  finally
  {
  }
}
catch(X e)
{
}
```
✓

(18)
```
try
{
}
catch(X e)
{
    try
    {
    }
    finally
    {
    }
}
```
✓

(19)
```
try
{
}
catch(X e)
{
    finally
    {
    }
}
```
CE: finally without try.

(20)
```
try
{
}
catch(X e)
{
}
finally
{
    try
    {
    }
    catch(X e)
    {
    }
}
```
✓

(21)
```
try
{
}
catch(X e)
{
}
finally
{
    finally
    {
    }
}
```
CE: finally without try.

(22)
```
try
{
}
catch(X e)
{
}
finally
{
}
finally
{
}
```
CE: finally without try.

(23)
```
try
    sopln("try");
catch(X e)
{
    sopln("catch");
}
finally
{
}
```
✗

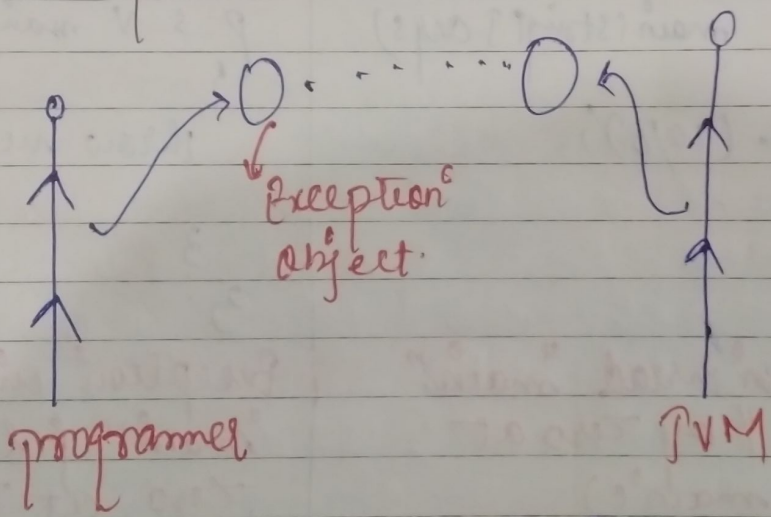
(24)
```
try
{
3
catch(X e)
   sopln("catch");
finally
{
3
```
X

(25)
```
try
{
3
catch(X e)
{
3
finally
   sopln("finally");
```
X

11, 12 points are also covered above.

Lecture 75 :- (14) Throw & Throws :-

14. Throw keyword :-



Some times we can create Exception Object explicitly. we can handover to the JVM manually for this we have to use throw keyword.

```
throw new AE("/ by zero");
```

→ Creation of Arithmetic object explicitly.

throw → handover our created exception object to the JVM manually.

Hence, the main objective of throw keyword is to handover our created exception object to the JVM manually.

Hence the result of following two programs is exactly same.

```
class Test
{
  p s v main(String[] args)
  {
    sopln(10/0);
  }
}
```

Exception in thread "main" j.l.AE: / by zero at Test.main()

In this case main() method is responsible to create exception object and handover to the JVM.

```
class Test
{
  p s v main(String[] args)
  {
    throw new AE("/ by zero");
  }
}
```

Exception in thread "main" j.l.A.E: / by zero at Test.main()

In this case programmer creating Exception object explicitly and handover to the JVM manually.

Best use of throw keyword is for user define exceptions or customized Exception.

```
eg: withdraw(double amount)
    {
        if(amount > balance)
            throw new InsufficientFundException();
```

Case 1:

```
throw e;
```

If e refers null then we will get (NPE).

```
class Test
{
    static AE e = new AE();
    p s v main(String[] args)
    {
        throw e;
    }
}
```

RE: AE

```
class Test
{
    static AE e;
    p s v main(String[] args)
    {
        throw e;
    }
}
```

RE: NPE

Case 2:- After throw statement we are not allowed to write any statement directly otherwise, we will get CTE: Unreachable statement.

```
class Test
{
    p s v main(String[] args)
    {
```

```
    sopln(10/0);
    sopln("Hello");
  }
}
```

RE: AE: / by zero.

```
class Test
{
  p s v main(String[] args)
  {
    throw new AE("/ by zero");
    sopln("Hello");
  }
}
```

CE: Unreachable statement.

Case 3: We can use throw keyword only for throwable types. If we are trying to use for normal Java objects we will get CTE: Incompatible types.

```
class Test
{
  p s v main(String[] args)
  {
    throw new Test();
  }
}
```

CE: incompatible types
found: Test
req: j.l.Throwable

```
class Test
{
  p s v main(String[] args)
  {
     throw new Test();
  }
}
```

RE: exception in thread "main" Test at Test.main().

Lecture 76 :(15) Throws Keyword:

In our program if there is a possibility of rising checked exception then compulsory we should handle that checked exception. Otherwise we will get CTE: Unreported exception xxxx must be caught or declared to be thrown.

eg1:

```
import java.io.*;
class Test
{
  p s v main(String[] args)
  {
     PrintWriter pw = new PrintWriter("abc.txt");
     pw.println("hello");
  }
}
```

CE: "Unreported exception java.io.FileNotFoundException must be caught or declared to be thrown.

Eg2:
```
class Test
{
    p s v main(String[] args)
    {
        Thread.sleep(10000);
    }
}
```
CE: Unreported Exception java.lang.InterruptedException must be caught or declared to be thrown.

We can handle this CTE by using the following two ways.

1- Way: By using try-catch.
```
class Test
{
    p s v main(String[] args)
    {
        try
        {
            Thread.sleep(10000);
        }
        catch(InterruptedException e)
        {
        }
    }
}
```
2- By using throws keyword:

We can use

throws Keywords to delegate responsibility of exception handling to the caller, [caller may be another method or JVM] then caller method is responsible to handle that exception.

eg:- class Test
{
    p s v main(String[] args) throws InterruptedException
    {
        Thread.sleep(10000);
    }
}

Conclusion of throws Keyword :-

- throws Keyword required only for checked exception and use of throws Keyword for unchecked exceptions there is no use or impact.
- throws Keyword required only to convince compiler and use of throws keyword doesn't prevent abnormal termination of the program.

class Test
{
    p s v main(String[] args) throws IE.
    {
        dostuff();
    }
    p s v dostuff() throws IE
    {
        doMorestuff();
    }

```
P s v doMoreStuff() throws IE
{
    Thread.sleep(10000);
}
}
```

CE: unreported exception j.l.IE must be caught or declared to be thrown.

eg: diff levels of delegation.

In the above program if we remove atleast one throws statement then the code won't compile.

throws clause

1. we can use to delegate responsibility of exception handling to the caller (It may be method or JVM)

2. It is required only for checked exceptions and usage of throws keywords for unchecked exceptions there is no impact.

3. It is required only to convience compiler and usage of throws does not prevent abnormal termination of program.

Note: It is recommended to use try-catch over the throws keyword.

Case

Case 2

Case 1°: We can use throws keyword for methods and constructor but not for classes.

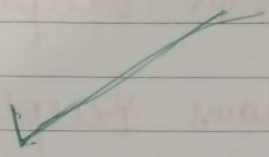
```
class Test throws Exception°   ✗
{
  Test() throws Exception°     ✓
  {
  }
  public void m1() throws Exception°.   ✓
  {
  }
}
```

Case 2°: We can use throws keyword only for throwable types. If we are trying to use for normal java classes. then we will get CTE: Incompatible types
found: Test
req: j.l.Throwable.

```
class Test
{
  public void m1() throws Test
  {
  }
}
```

CE: incompatible types
found: Test
req: j.l.Throwable.

```
class Test extends RuntimeException
{
  public void m1() throws Test
  {
  }
}
```



Case 3:-

```
class Test
{
  p s v main(String[] args)
  {
    throw new Exception();
  }
}
```

Checked

CE: Unreported exception j.l.Exception must be caught or declared to be thrown.

```
class Test
{
  p s v main(String[] args)
  {
    throw new Error();
  }
}
```

Unchecked

RE: Exception in thread "main" j.l.Error
at Test.main().

Note:- In our with in the try block if there is no chance of raising an exception then we can't write catch block for that exception, otherwise we will get CTE: Exception 'XXXX' is never thrown in body of corresponding try statement.
But this rule is applicable only for fully checked exceptions.

1
```
class Test
{
  p s v main(String[] args)
  {
    try
    {
      sopln("Hello");
    }
    catch(AE e)
    {
    }
  }
}
```
unchecked

Op:- Hello.

```
class Test
{
  p s v main(strm)
  {
    try
    {
      sopln("Hello");
    }
    catch(Exception e)
    {
    }
  }
}
```
partially checked

Op:- Hello.

3)
```
import java.io.*;
class Test
{
  p s v main(strm)
  {
    try
    {
      sopln("Hello");
    }
    catch(IoException e)
    {
    }
  }
}
```

CE: exception java.io.IoException is never thrown in body of corresponding try statement.

```
class Test
{
  p s v main(Strm)
  {
    try
    {
      Sop1n("Hello");
    }
    catch(InterruptedException e)
    {
    }
  }
```

Fully checked.

CE: exception 'j.l.IE is never thrown in body of corresponding try statement.

```
class Test
{
  p s v main(Strm)
  {
    try
    {
      Sop1n("Hello");
    }
    catch(Error e)
    {
    }
  }
}
```

Unchecked

O/p: Hello.

Lecture - 77 : Customized Exception.

16. Exception handling Keywords Summary :

① try → To maintain Risky Code.

② Catch → to maintain exception handling code

③ finally → to maintain cleanup Code.

④ throw → to hand-over our created exception object to the JVM manually.

⑤ throws → to delegate responsibility of exception handling to the caller.

17. Various possible CTE in Exception Handling :

1. unreported exception XXXX ; must be caught or declared to be thrown.
2. Exception XXXX has already been caught.
3. Exception XXXX is never thrown in body of corresponding try statement.
4. unreachable statement.
5. Incompatible Types.
   found : Test
   req : java.lang.Throwable.
6. try without catch or finally.
7. Catch without try.
8. finally without try.

18. Customised or User defined exceptions:

Sometimes to meet programming requirements we can define our own exceptions. Such type of exceptions are called Customized or user defined exceptions.

Eg: TooYoungException
TooOldException.
InSufficientFundsException.

Eg:

```
class TooYoungException extends RuntimeException
{
    TooYoungException(String s)
    {
        super(s);
    }
}
class TooOldException extends RuntimeException
{
    TooOldException(String s)
    {
        super(s); // To make description available to default exception Handler.
    }
}
class CustException
{
    public s v main(string[] args)
    {
```

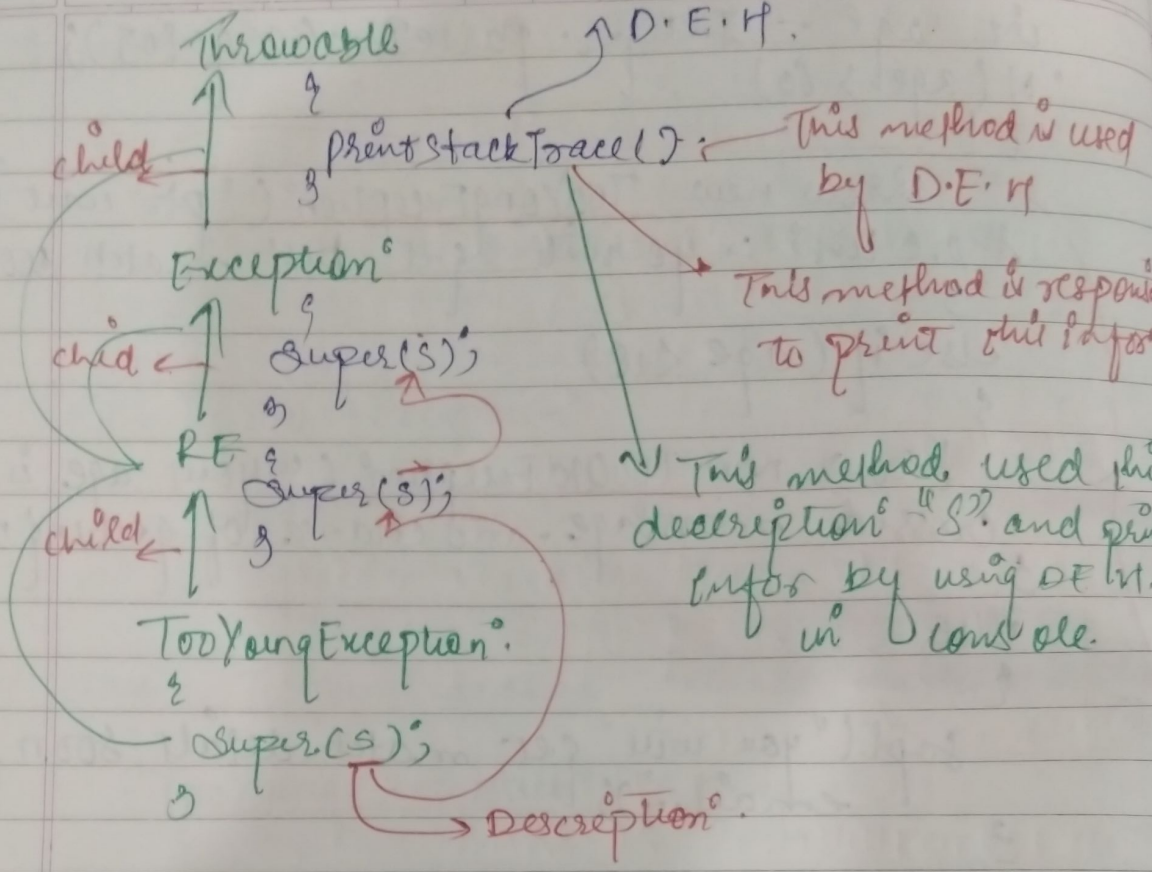
Defining Customised exception

NOTE: 1)

```
int age = Integer.parseInt(args[0]);
if(age>60)
{
  throw new TooYongException("plz wait some more time ... u will get best match soon");
}
else if(age<18)
{
  throw new TooOldException("your age is already crossed marriage.. no chance of getting marriage");
}
else
{
  sopl("you will get match details soon by email...");
}
}
}
```

Note:- 1) throw keyword is best suitable for user defined or customized exception but not for pre-defined exceptions.

2) It is highly recommended to define customized exception as unchecked. i.e. we have to extends runtime exception but not Exception.



Sol

Lecture 78 (19) Top 10 - Exception.

1. Based on the person who is rising an exception all exceptions are divided into two categories.

1. JVM Exceptions
2. programmatic Exceptions.

1. JVM Exception :- The Exceptions which are rised automatically by JVM whenever a particular event occurs

are called JVM exceptions. eg:- ArithmeticException, NullPointerException etc.

2. programmatic Exceptions:- The exceptions which are rised explicitly either by programmer or by API developer to indicate that some-thing goes wrong. are called programmatic exceptions. eg:- i) TooOldException ii) IllegalArgu-mentException etc.

```
class Thread
{
public final void setPriority(int newPriority)
{
    if(newPriority > MAX_PRIORITY || newPriority <
      MIN_PRIORITY){
      throw new IllegalArgumentException();
    }
}
```

```
Thread t = new Thread();
t.setPriority(5);
t.setPriority(15); RE: IllegalArgumentException
```

To-do imp Exception in Java:-

i. ArrayIndexOutOfBoundException:- It is the child class of RuntimeException and it is unchecked.
ii) Rised automatically by JVM hence whenever we are trying to access array element with OutOfRange index.

eg: int[] x = new int[4] [0 to 3]

Sopln(x[0]); ✓

Sopln(x[10]); RE: ArrayIndexOutOfBoundsException
Sopln(x[-10]);

2. NullPointerException :- It is the child class of RuntimeException and hence it is unchecked.

(ii) Rised automatically by JVM whenever we are trying to perform any operation on Null:

String s = null;
Sopln(s.length()); RE: NullPointerException

3. ClassCastException :- It is the child class of RuntimeException and hence it is unchecked.

(i) String s = new String("durga");
Object o = (Object)s; ✓

(ii) Object o = new Object(); → parent object.
String s = (String)o; → RE: ClassCastException

Rised automatically by JVM whenever we are trying to typecast parent object to

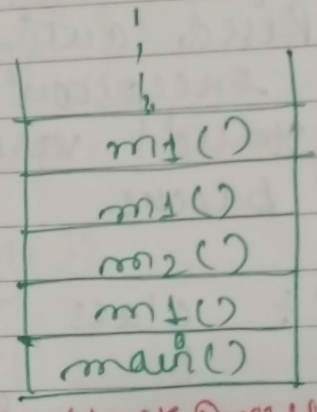
child type.

Object o = new String("durga"); ✓
↳ child object

String s = (String) o;

4. StackOverflowError :- It is the child class of error and hence it is unchecked. Rised automatically by JVM. When new we are trying to perform recursive method call.

```
class Test
{
    p s v m1()
    {
        m2();
    }
    p s v m2()
    {
        m1();
    }
    p s v main(String[] args)
    {
        m1();
    }
}
```



RE: StackOverflowError.

6. NoClassDefFoundError :- It is the child class of error and hence it is unchecked.

Rised automatically by JVM whenever ~~we are~~ JVM

unable to find required .class file.

eg:- Java Test

If Test.class file is not available then we will get Runtime Exception:
NoClassDefFoundError: Test.

6. ExceptionInInitializerError :- It is the child class of Error, hence it is unchecked.

Rised automatically by JVM if any exception occurs while executing static variable assignments and static blocks.

eg:- class Test
{
static int x = 10/0;
}

RE: ExceptionInInitializerError.
Caused by J.L.AE: / by zero.

class Test
{
static
{
String s = null;
sopln(s.length());
}
}

ExceptionInInitializerError caused by:
e.g. NPE.

Lecture 79:- try with resource & multicatch block:

7. IllegalArgumentException:- It is the child class of RuntimeException and hence it is unchecked.
Rised Explicitly either by programmer or by API developer to indicate that a method has been invoked with illegalArgument.

eg:- The valid range of Thread priorities is 1 to 10. If we are trying to set the priority with any other value then we will get runtimeException: IllegalArgumentException.

eg:- ~~Thread~~ t = new Thread();
t.setPriority(7); ✓
t.setPriority(15); RE: IllegalArgumentException.

8. NumberFormatException:- It is the direct child class of RuntimeException. / IllegalArgumentException which is the child class of RuntimeException, hence it is unchecked.
Rised explicitly by either by programmer or by API developer to indicate that we are trying to convert String to number, and the String is not properly formatted.

eg:- int i = Integer.ParseInt("10"); ✓
✗ int i = Integer.ParseInt("ten"); RE: NumberFormatException

1. **IllegalStateException** :- It is the child class of RuntimeException, hence it is unchecked. Raised explicitly either by programmer or by API developer to indicate that a method has been invoked at wrong time.

Eg:- After starting of a thread we are not allow to restart the same thread once again otherwise we will get RuntimeE: IllegalThreadStateException.

```
Thread t=new Thread();
t.start();
:
:
t.start(); X RE:
```

10. **AssertionError** :- It is the child class of error and hence it is unchecked. Raised explicitly by the programmer or by API developer to indicate that Assert statement fails.

Eg:- Assert (x>10);

RE: AssertionError

If x is not >10 then we will get RE:

Exception / Errors	Raised by
1) ArrayIndexOutOfBoundException. 2) NullPointerException. 3) ClassCastException. 4) StackOverflowErrors. 5) NoClassDefFoundErrors. 6) ExceptionInInitializerErrors.	• Raised automatically by JVM and hence these are JVM exceptions.
7) IllegalArgumentException. 8) NumberFormatException. 9) IllegalStartException. 10) AssertionException.	→ Raised explicitly either by programmer (or) by API developer and hence these are programatic Exception

20. 1.7 Version Enhancement wrt exception handling

As a part of 1.7 Vers^n in exception handling the following two concepts introduced

1. try with resources
2. Multi-catch block.

1. try with resources: Until 1.6V it is highly recommended to write finally Block to close

resources which are open as a part of try block.

1.6V

```
BR  br = null;
try
{
  br = new BR(new FR("input.txt"));
  // use br based on our requirement.
}
catch (IOException e)
{
  // Handling code
}
finally
{
   if (br != null)
   {
      br.close();
   }
}
```

The problems in this approch are:-

i) Compulsory programmer is required to close the resources inside finally block it increases complexity of programming.

ii) We have to write finally block compulsory & hence it increased length of the code and reduces readability.

→ To overcome above problems SUN people introduced try with resources in 1.7 version.

The main advantage of try with resources is whatever the resources we open as a part of try block will be closed automatically once control reaches end of try block either normally or abnormally. Hence we are not required to close explicitly. So that complexity of programing will be reduced.

We are not required to write finally block so that length of the code will be reduced & readability will be improved.

```
try( BR br = new BR(new FR("input.txt")))
                  ↳Resource
{
    // use br based on our requirement
    br will be closed automatically once control
    reaches end of try block either normally
    or abnormally. and we are not responsible
    to close explicitly.
}
Catch (IOException e)
{
    // Handling Code.
}
```

Conclusion :-

1) We can declare multiple resources but these resources should be seperated with ; semicolon.

try (R1; R2; R3)

→ {

}

eg:-

```
try(FileWriter fw=new FileWriter("output.txt");
    FR fr=new FR("input.txt");)
```

R1 — FileWriter fw=new FileWriter("output.txt");

R2 — FR fr=new FR("input.txt");)

(2) All resources should be autocloseable resources.

A resource is said to be autocloseable iff corresponding class implements java.lang.Autocloseable interface

All IO related resources, database related resources and network related resources are already implemented autocloseable interface. being a programer we are not required to do anything just we should aware the point

Autocloseable interface come in 1.7V, and it contains only once method close() / public void close()

```
try(R1;R2;R3)
{
}
```

≡

```
{
  R1.close();
  R2.close();
  R3.close();
}
```

→ Autocloseable
→ java.lang.Autocloseable
→ 1.7V
→ close().

(3) All resource reference variable are implicitly final, hence within the try block we can't perform reassignment. Otherwise, we will get CTE.

eg:
```
import java.io.*;
class TryWithResources
{
    P S V main(String[] args) throws Exception
    {
        try(BufferedReader br = new BufferedReader(new FileReader("input.txt")))
        {
            br = new BufferedReader(new FR("output.txt"));
        }
    }
}
```

CE: auto-closeable resource br may not be assigned.

(4) Until 1.6 V try should be associated with either catch or finally, but from 1.7 V onwards we can take only try with resource without catch or finally.

```
try(R)
{
}
```

the main advantage of try with resources is we are not required to write finally block explicitly because we are not required to close resources explicitly, hence multi 1.6V finally block is just like zero. but from 1.7V onwards it is dummy. and becomes zero.

2. Multi-catch block: Until 1.6V even though multiple different exceptions having same handling code. for every exception type we have to write a separate catch block. it increases length of the code and reduces readability.

eg:
```
try
{
:
}
catch (AE e)
{
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}
catch (NPE e)
{
    sopln(e.getMessage());
}
```

```
catch (InterruptedException e)
{
    sopln(e.getMessage());
}
```

B.
To overcome this problem SUN people introduced multicatch block in JDK 1.7V.
According to this, we can write a single catch-block that can handle multiple different type of exceptions.

```
try
{
=
}
catch (AE/IOException e)
{
    e.printStackTrace();
}
catch (NPE/InterruptedException e)
{
    sopln(e.getMessage());
}
```

The main advantage of this above approach length of the code will be reduced and readability will be improved.

eg:-

```
import java.io.*;
class MultiCatchBlock
```

```
{
  p s v main(String[] args)
  {
    try
    {
      sopln(10/0);
      String s = null;
      sopln(s.length());
    }
    catch(ArithmeticException | NPE e)
    {
      sopln(e);
    }
  }
}
```

In the above example whether raised exception is either AE or NPE the same catch block can listen/respond.

In multi-catch block there should not be any relation b/w exception types (either child to parent or parent to child or same type) otherwise we will get CTE.

```
try
{
}
catch(AE | Exception e)
{
```

e.printStackTrace();
}

CF: Alternatives in a multi-catch statement cannot be related by subclassing.

* Exception propagation :- Inside a method if an exception is raised and if we are not handling that exception, then exception object will be propagated to caller then caller method is responsible to handle exception. this process is called exception propagation.

```
m1()
{
   m2();
}

m2()
{
   sop(10/0);
}
```

* Rethrowing exception :- We can use this approach to convert one exception type to another exception type.

eg:
```
try
{
   sop(10/0);
}
catch(AE e)
{
   throw new NullPointerException();
}
```

End of Exception Handling